Code Analysis

Steven Lavenhar, Cigital, Inc. [vita[3]]

2006-02-22; Updated 2008-11-03 L3 / L, M[4]

Software security is first and foremost about identifying and managing risks. One of the most effective ways to identify and manage risk for an application is to iteratively review its code throughout the development cycle. Substantial net improvements in software security can be realized through the formal use of design and code inspection.

Introduction

Software security is about building secure software: designing software to be secure, making sure that software is secure, and educating software developers, architects, and users about how to build secure applications. Developing robust, enterprise-level applications is a difficult task, and making them completely secure is virtually impossible. Too often software development organizations place functionality, schedules, and costs at the forefront of their concerns, and make security and quality an afterthought. Nearly all attacks on software applications have one fundamental cause: the code is not secure due to defects in its design, implementation, testing, and operations. Software security is first and foremost about identifying and managing risks. One of the most effective ways to identify and manage risk for an application is to iteratively review its code throughout the development cycle. Application security is the key risk area for exploits, and exploits of applications can be devastating.

A vulnerability is an error that an attacker can exploit. Many types of vulnerabilities exist in software systems, including local implementation errors, interprocedural interface errors (such as a race condition between an access control check and a file operation), design-level mistakes (such as error handling and recovery systems that fail in an insecure fashion), and object-sharing systems that mistakenly include transitive trust issues [McGraw 04[7]]. Vulnerabilities typically fall into two categories: bugs at the implementation level and flaws at the design level.

This document focuses on implementation-level security issues; these vulnerabilities are the target of the source-code analyst. Design-level flaws, which are also an important part of the big picture, are discussed elsewhere in the BSI portal.

Vulnerabilities exist within applications because of programmatic weaknesses. The following classes of vulnerabilities are common:

- **Race Conditions**. Race conditions take on many forms but can be characterized as scheduling dependencies between multiple threads that are not properly synchronized and cause an undesirable timing of events. An example of a race condition that could have a negative outcome on security is when a specific sequence of events is required between Event A and Event B, but a race occurs and the proper sequence is not ensured by the program. There are a number of programming constructs that can be used to control the synchronization of threads such as semaphores, mutexs, and critical sections. In some application frameworks, such as .NET, there are thread management classes that can be used to handle thread synchronization and scheduling. Race conditions fall into three main categories:
  - indefinite loops, which cause a program to never terminate or never return from some flow of logic or control
  - deadlocks, which occur when the program is waiting on a resource without some mechanism for timeout or expiration and the resource or lock is never released

3. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/197-BSI.html (Lavenhar, Steven)
7. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/code/213-BSI.html#dsy213-BSI_mcgraw04 (Code Analysis - References)

    - resource collisions, which represent failures to synchronize access to shared resources, often resulting in resource corruption or privilege escalations (see [Bishop 96[14]])
- **Input Validation**. Trusting user and parameter input can lead to security problems. Vulnerabilities such as semantic or SQL injection can pose risks to confidentiality and integrity. Semantic injection occurs when a well-formed user input is processed and produces unexpected results. SQL injection is similar to semantic injection in that a well-formed SQL syntax is passed on to a database server for processing. SQL injection attacks represent a serious threat to any database-driven site. The primary goal of a SQL injection attack is to attempt to manipulate a query or information sent to a SQL backend to gain control of that SQL server. The methods behind a SQL injection attack are easy to learn and the resulting damage caused by such an attack can be considerable. Numerous white papers and other references are available on the Internet that document how to perform such attacks (also see [href="daisy:213-BSI#howard02"Howard 02]). Despite the risks of these attacks a significant number of systems on the Internet are vulnerable to this form of attack.
- **Exceptions**. Exceptions are events that disrupt the normal flow of code. Proper exception handling provides built-in support for handling anomalous situations (exceptions) which may occur during the execution of a program. With exception handling, a program can communicate unexpected events to a higher execution context that is better able to recover from such abnormal events. These exceptions are handled by code that is outside the normal flow of control. Exceptions can be avoided by testing for conditions that can lead to an exception. Languages such as Java and C++ provide exception handling through *try and catch* code blocks. In the C++ exception handler example below, the compound-statement that follows the try clause is a guarded section of code. The throw-expression throws an exception. The compound-statement that follows the catch clause is the exception handler, and catches the exception thrown by the throw-expression. The exception-declaration statement that follows the catch clause indicates the type of exception the clause handles. The type can be any valid data type, including a C++ class.

  try-block :

  try compound-statement handler-list

  handler-list :

  handler handler-list

  handler :

  catch (exception-declaration) compound-statement

  exception-declaration :

  type-specifier-list declarator

  type-specifier-list abstract-declarator

  type-specifier-list

  ...

  throw-expression :

  throw assignment-expression

  If the exception-declaration statement is an ellipsis (...), the catch clause handles any type of exception, including system-generated and application-generated exceptions. This includes exceptions such as memory protection, divide-by-zero, and floating-point violations. An ellipsis catch handler must be the last handler for its try block. The operand of the throw is syntactically similar to the operand of a return statement.

  When an exception occurs inside a try block, such as a read command to a missing file, an exception is thrown and caught by a catch block designed to catch that specific exception. If there are no catch blocks designed to catch the exception, the program risks crashing or instability. Some other security concerns that arise from exception handling are discussed in [McGraw 03[32]] and [Seacord 08[33]].

- **SQL Injection**. SQL injection is a technique used by attackers to take advantage of non-validated input vulnerabilities to pass SQL commands through a Web application for execution by a backend database. Attackers take advantage of the fact that programmers often chain together SQL commands with user-provided parameters, and the attackers, therefore, can embed SQL commands inside these parameters (see [SPI 02[34]]). The result is that the attacker can execute arbitrary SQL queries and/or commands on the backend database server through the Web application.

  Typically, Web applications use string queries, where the string contains both the query itself and its parameters. The string is built using server-side script languages such as ASP or JSP and is then sent to the database server as a single SQL statement. The following example demonstrates an ASP code that generates a SQL query:

  ```
  sql_query= "SELECT ProductName, ProductDescription FROM Products
  WHERE ProductNumber " & Request.QueryString("ProductID")
  ```

  The corresponding SQL query is executed:

  ```
  SELECT ProductName, ProductDescription FROM Products
  WHERE ProductNumber = 123
  ```

  An attacker may abuse the fact that the ProductID parameter is passed to the database without sufficient validation. The attacker can manipulate the parameter's value to build malicious SQL statements.

  For example, setting the value 123 OR 1=1 to the ProductID variable results in the following SQL Statement:

  ```
  SELECT ProductName, Product Description From Products
  WHERE ProductNumber = 123 OR 1=1
  ```

  This condition would always be true and all ProductName and ProductDescription pairs are returned. The security model used by many Web applications assumes that a SQL query is a trusted command. This enables attackers to exploit SQL queries to circumvent access controls, authentication, and authorization checks. In some instances, SQL queries may allow access to host operating system level commands. This can be done using stored procedures. For example, the Microsoft SQL Server extended stored procedure xp_cmdshell executes operating system commands. Using the same example, the attacker can set the value of ProductID to be 123;EXEC master..xp_cmdshell dir--, which returns the list of all the files in the current directory of the SQL Server process.

- **Buffer Overflows**. Buffer overflows are the principal method used to exploit software by remotely injecting malicious code into a target [Seacord 05[43], Viega 01[44]]. The root cause of buffer overflow problems is that C and C++ are inherently unsafe. There are no bounds checks on array and pointer references, meaning a developer has to check the bounds (an activity that is often ignored) or risk encountering problems. A number of unsafe string operations also exist in the standard C library, such as the notorious strcpy() function.

  The C and C++ programming languages allow programmers to create storage at runtime in two different sections of memory: the stack and the heap. Generally, heap-allocated data are the kind you get when you malloc() or new something. Stack-allocated data usually include non-static local variables and any parameters passed by value. When writing to buffers, C/C++ programmers must take care not to store more data in the buffer than it can hold. When a program writes past the bounds of a buffer a buffer overflow occurs. When this happens, the next contiguous chunk of memory is overwritten. Since C and C++ are inherently unsafe, they allow programs to overflow buffers at will. There are no runtime checks that prevent writing past the end of a buffer, so programmers have to perform the checks in their own code, or they will risk running into problems in the future.

  Reading or writing past the end of a buffer can cause a number of diverse (and often unanticipated) behaviors: (*a*) programs can act in strange ways, (*b*) programs can fail completely, and (*c*) programs can proceed without any noticeable difference in execution. The side effects of overrunning a buffer depend on the following:

  - how much data are written past the buffer bounds

- what data (if any) are overwritten when the buffer gets full and spills over

- whether the program attempts to read data that are overwritten during the overflow

- what data end up replacing the memory that gets overwritten

The indeterminate behavior of programs that have overrun a buffer makes them particularly tricky to debug. In the worst cases, a program may be overflowing a buffer and not showing any adverse side effects at all. As a result, buffer overflow problems are often invisible during standard testing. The important thing to realize about buffer overflows is that any data that happen to be allocated near the buffer can potentially be modified when the overflow occurs.

- The most common form of buffer overflow, called the *stack overflow* [Aleph 96[52]], can be easily prevented. Stack-smashing attacks target a specific programming fault: the careless use of data buffers allocated on the program's runtime stack. An attacker can take advantage of a buffer overflow vulnerability by stack-smashing and running arbitrary code, such as code that invokes a shell in such a way that control gets passed to the attack code. More esoteric forms of memory corruption, including the heap overflow, are harder to avoid. By and large, memory usage vulnerabilities will continue to be a fruitful resource for exploiting software until modern languages that incorporate modern memory management schemes are in wider use.

  Integer Overflows. An integer overflow occurs when an attempt is made to place an integer value within a storage space that is not large enough to contain the binary representation for that integer. Integer overflow vulnerabilities can be prevented by using “safe integer operations[53].”

A best practices approach to developing software can help to reduce exploitable vulnerabilities substantially. These best practices include the following:

- performing bounds checking
- checking configuration files
- checking command-line parameters
- checking environment variables
- setting initial values for data
- monitoring logs
- implementing file integrity solutions
- using stack protection
- training your developers
- reviewing source code
- performing third party audits

Source Code Review

Source code review for security, along with architectural risk analysis, ranks very high on the list of software security best practices. Substantial net improvements in software security can be realized through the formal use of design and code inspection. Peer review of source code is meant to be simple, informal, and easily integrated into the regular development processes. Reviewers meet one-on-one with developers and review code visually to determine if it meets well-known development criteria. Reviewers consider coding standards and use best practices code review checklists that list features like comments, documentation, the unit test plan, and the code’s compliance with requirements. Developers present unit test plans for the code as part of the review and detail how the code that they have written can be tested. The test plan includes a test procedure, inputs, and expected outputs. Requirements and specifications are critical to a peer review because they largely determine whether or not the code has implemented the functionality correctly.

There are several methodologies for conducting source code reviews. In the Top-Down approach, the auditors examine the source code for certain types of vulnerabilities without needing to have an in-depth understanding of the specifics of how a program functions. This approach is useful for determining certain

types of architectural and design flaws, but it has limitations. Any vulnerability requiring a deep knowledge of program architecture and design likely will be missed. This can be addressed by the Bottom-Up approach to auditing code, which is based on the premise that the auditors have a deep understanding of the code. While this approach is very time consuming, it allows the auditors to identify vulnerabilities that may be very subtle or involve very complex programmatic interactions. The Selective approach is a compromise between Top-Down and Bottom-Up analysis. In this approach the audit is focused on code which is likely to be reached by attackers. The choice of methodology often is dependent on the goals of the analysis and the specific types of vulnerabilities being examined.

Manual auditing of code for security vulnerabilities can be very time consuming. In addition, to perform a manual analysis effectively, the code auditors must first know what security vulnerabilities look like before they can rigorously examine the code. Static analysis tools compare favorably to manual audits because they are faster, can be utilized to evaluate programs much more frequently, and can encapsulate security knowledge in a way that does not require the tool operator to have the same level of security expertise as a human auditor. On the other hand, these tools cannot replace a human analyst; they can only speed up those tasks that are easily automated.

White Box and Black Box Code Analysis Tools

Both white and black box testing methods can be used to identify software vulnerabilities. These two methods use different approaches depending on whether the tester has access to source code. White box testing involves analyzing source code and is very effective in finding programming errors. Automated scanning tools are often used in white box analysis to identify bugs in code.

Static source code analysis (see [Chess 04[72]] and [Chess 07[73]]) is the process by which software developers check their code for problems and inconsistencies before compiling. Software development organizations can automate the analysis of source code by utilizing tools that automatically analyze entire programs. Static analysis tools scan the source code and automatically detect errors that typically pass through compilers and become latent problems. Examples of problems detected by static code analyzers include the following:

- syntax problems
- unreachable code
- unconditional branches into loops
- undeclared variables
- uninitialised variables
- parameter type mismatches
- uncalled functions and procedures
- variables used before initialization
- non-usage of function results
- possible array bound errors
- misuse of pointers

Many modern static analysis tools generate reports that graphically present the analysis results and recommend potential resolutions to identified problems.

Static code analysis discovers subtle and elusive implementation errors before they reach testing or fielded system status. By correcting subtle errors in the code early, software development organizations can save engineering costs in testing and long-term maintenance. Static code analysis tools can be applied in a variety of different ways, all of which lead to higher quality in software.

72. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/code/213-BSI.html#dsy213-BSI_chess04 (Code Analysis - References)
73. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/code/213-BSI.html#dsy213-BSI_chess07 (Code Analysis - References)

Testing for security vulnerabilities is complicated by the fact that they often exist in hard-to-reach states or crop up in unusual circumstances. Static analysis has the advantage of being able to be applied before a program reaches a level of completion at which dynamic analysis or other types of testing can be meaningfully performed. However, static code analyzers should not be viewed as a panacea. Static analysis tools look for a fixed set of patterns, or rules, in the code in a manner similar to virus checking programs. While some of the more advanced tools available allow new rules to be added to the rulebase, the tool will never find a problem if a rule has not been written yet for it. These tools also can produce false positives and false negatives. Results that indicate that zero security defects were found should not be taken to mean that your code is secure; these results mean that your code has none of the existing patterns in the program's rulebase that represent security defects.

The principal promise of static analysis tools is that they can identify many common coding problems automatically. However, implementation flaws due to programmer errors are only part of the problem. Static analysis tools cannot evaluate design and architecture flaws. They cannot identify poorly designed crypto libraries or improperly selected algorithms and cannot identify when there are design problems that cause confusion between authentication and authorization. They also cannot even identify passwords or magic numbers embedded in code. One further drawback to using automated scanning is that the tools are prone to false positives when a potential vulnerability does not exist. This is especially true of older freeware tools, most of which are not actively supported; many analysts do not find these tools to be useful when analyzing real-world software systems. Commercial tool vendors are actively addressing the problem of false positives and have made considerable progress since the early days of security scanning, but much remains to be done. Security analysis tools and their false alarms are discussed in detail elsewhere on the BSI portal.

While they are extremely useful, static analysis tools can only identify a subset of the vulnerabilities leading to security problems. Static analysis tools must be used in conjunction with manual auditing and other software assurance methods to reduce vulnerabilities that are not amenable to being identified by patterns and rules. Nevertheless, using static analysis methods is a good technique for analyzing certain kinds of software.

Metrics Analysis

Metrics analysis looks at a quantitative measure of the degree to which the code under consideration possesses a given attribute. An attribute is a characteristic or a property of the code. The process of using code metrics begins by deriving the metrics that are appropriate for the code under consideration. Then data are collected and metrics are computed. The metrics are computed and compared to pre-established guidelines and historical data. The results of these comparisons are used to guide modifications to the code to improve the corresponding code qualities.

There are two distinct classes of quantitative software metrics, absolute and relative. Absolute metrics are numerical values that represent a characteristic of the code, such as the probability of failure metric, the number of references to a particular variable in an application, or the number of lines of code. Absolute metrics, such as the number of lines of code, do not involve uncertainty. There can be one and only one correct numerical representation of a given absolute metric. In contrast, relative metrics are a numeric representation of an attribute that cannot be directly measured, such as the degree of difficulty in testing for buffer overflows. There is no objective, absolute way to measure such an attribute. Multiple variables are factored into an estimation of the degree of testing difficulty, and any numeric representation is only an approximation.

Metrics provide management with critical insight into how to support strategic decision making processes. When considered separately, a metric such as the number of defects per 1000 lines of code provides very little business meaning. However, a metric such as the number of security breaches per 1000 lines of code provides a much more useful and interesting relative value. It can be used to compare and contrast a given system's security defect density against similarly sized systems and thus provide management with useful data for decision making.

Code Analysis Tools

The Source Code Analysis Tools[94] content area provides a discussion of tools for evaluating security vulnerabilities in source code. Code samples are provided to run tools against to verify that the tools are able to detect known problems in the code.

Code Analysis References

A list of references[95] for the Code Analysis content area.

Secure Coding Sites

For some resources about secure coding in addition to what is provided on the Build Security In website, see the BSI Secure Coding Sites [96]page.

Code Analysis Process Diagrams

Cryptanalysis[97]
Dynamic Analysis[98]
Fault Injection[99]
Metric Analysis[100]
Random Number Generator Analysis[101]
Source Code Review[102]
Static Code Analysis[103]

Cigital, Inc. Copyright

94. http://buildsecurityin.us-cert.gov/bsi/articles/tools/code.html (Source Code Analysis)
95. http://buildsecurityin.us-cert.gov/bsi/articles/best-practices/code/213-BSI.html (Code Analysis - References)
96. http://buildsecurityin.us-cert.gov/bsi/securecoding.html (Secure Coding Sites)
97. http://buildsecurityin.us-cert.gov/bsi/468-BSI.html (Cryptanalysis Process Diagram)
98. http://buildsecurityin.us-cert.gov/bsi/469-BSI.html (Dynamic Analysis Process Diagram)
99. http://buildsecurityin.us-cert.gov/bsi/470-BSI.html (Fault Injection Process Diagram)
100. http://buildsecurityin.us-cert.gov/bsi/471-BSI.html (Metric Analysis Process Diagram)
101. http://buildsecurityin.us-cert.gov/bsi/472-BSI.html (Random Number Generator Analysis)
102. http://buildsecurityin.us-cert.gov/bsi/473-BSI.html (Source Code Review)
103. http://buildsecurityin.us-cert.gov/bsi/474-BSI.html (Static Code Analysis)
1. mailto:copyright@cigital.com